Multiprocessor cache coherence

The observation that the use of large, multilevel caches can greatly reduce the memory bandwidth demands of a processor is the key insight that motivated centralised memory multiprocessors.

With recent, high-performance processors, the memory demands have outstripped the capability of reasonable buses and some microprocessors directly connect memory to a single chip, sometimes called *backside* or *memory bus*, to distinguish it from the bus used to connect to I/O.

Accessing a chip's local memory, whether for an I/O operation or for an access from another chip, requires going through the chip that "owns" that memory. Thus, access to memory is **asymmetric**: faster to the local memory and slower to the remote memory. In a multicore, that memory is shared among all cores on a single chip, but the asymmetric access to the memory of one multicore from the memory of another remains.

Symmetric shared-memory machines usually support the caching of both **shared and private** data: private data are used by a single processor, while shared data are used by multiple processors, essentially providing communication among the processors through reads and writes of the shared data.

When shared data is cached, the shared value may be replicated in **multiple caches**. In addition to the reduction in access latency and required memory bandwidth, this replication also provides a reduction in contention that may exist for shared data items that are being read by multiple processors simultaneously.

Caching of shared data, however, introduces a new problem: **cache coherence**: the view of memory held by two different processors is through their individual caches which, without ant additional precautions, could end up seeing two different values. This difficulty is generally referred to as the **cache coherence problem**. This problem exists because we have both a global state, defined primarily by the main memory, and a local state, defined by the individual processor caches, which are private to each processor core.

Coherence and consistency in caches

Coherence property

Informally, we could say that a memory system is **coherent** if any read of a data item returns the most recently written value of that data item. This definition, however, is simplistic; the reality of things is much more complex. This simple definition contains two different aspects of a memory system behaviour, both of which are critical to writing correct shared memory programs: the first aspect, *coherence*, defines what values can be returned by a read; the second aspect, *consistency*, determines when a written value will be returned by a read.

We can formally define coherence like so:

Coherence

A memory system is **coherent** if:

1. A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always return the value written by P.
2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.
3. Writes to the same location are *serialised*: two writes to the same location by any two processors are seen in the same order by all processors.

The first property simply preserves program order. The second property defines the notion of what it means to have a coherent view of memory.

The need for write serialisation is more subtle: suppose we did not serialise writes and processor P_1 writes location X followed by P_2 writing location X. Serialising the writes ensures that every processor will see the write done by P_2 at some point. If we did not serialise the writes, it might be the case that some processors could see the write of P_2 first and then the write of P_1, maintaining the value written by P_1 indefinitely. The simplest way to avoid such difficulties is to ensure that all writes to the same location are seen in the same order; this property is called *write serialisation*.

Consistency property

Although the three properties just described are sufficient to ensure coherence, the question of when a written value will be seen is also important: we cannot require

that a read of X instantaneously see the value written for X by some other processor. If, for example, a write of X on one processor precedes a read of X on another processor by a very small time, it may be impossible to ensure that the read returns the value of the data written, since the written data may not even have left the processor at that point.

The issue of exactly *when* a written value must be seen by a reader is defined by a **memory consistency model**—a topic we will focus on later.

Coherence protocols

Unlike I/O, where multiple data copies are a rare event, a program running on multiple processors will normally have copies of the same data in several caches. In a *coherent* multiprocessor, the caches provide both **migration** and **replication** of shared data items:

- Migration is the process through which a data item is moved to a local cache and used there in a transparent fashion by the processor. This reduces both latency to access the data item and the bandwidth demand on the shared memory.
- Replication is the process through which a data item is copied across caches in order to enable the simultaneous read of such data item.

The protocols to maintain coherence for multiple processors are called **cache coherence protocols**. The key to implementing one is to track the state of any sharing of a data block.

There are two main techniques to enforce coherence in a multiprocessor: directory-based protocols and snooping protocols.

In **directory-based** protocols, the sharing status of a particular block of physical memory is kept in one location called the *directory*. There are two very different types of directory-based cache coherence: in an SMP, we can use one *centralised directory*; in a DSM, it makes no sense to have a single directory, since it would create a single point of contention and make it difficult to scale to many multicore chips, so distributed directories are used—although they are much more complex.

In **snooping** protocols, rather than keeping the state of sharing in a single directory, every cache that has a copy of the data from a block of physical memory could track the sharing status of one block. In an SMP, the caches are typically all accessible via

some broadcast medium and all cache controllers *snoop* on the medium to determine whether or not they have a copy of a block that is requested.

Snooping protocols became popular with multiprocessors using single-core microprocessors and caches attached to a single shared memory by a bus. The bus provided a convenient broadcast medium to implement the snooping protocols.

Multicore architectures changed the picture significantly, since all multicores share some level of cache on the chip. Thus, some designs switched to using directory protocols, since the overhead was small.

In this course, we focus on snooping protocols, thus only those will be described here.

Snooping protocols

There are two ways to maintain the coherence requirement described earlier: one method is to ensure that a processor has exclusive access to a data item before it writes that item. This style of protocol is called **write invalidate protocol**, since it invalidates other copies on a write. This is by far the most common protocol applied to implement snooping protocols.

Let's make an example: consider a write followed by a read by another processor. Since the write requires exclusive access, any copy held by the reading processor **must be invalidated**. Thus, when the read occurs, **it misses** in the cache and is **forced to fetch a new copy** of the data. If two processors attempt to write the same data simultaneously, **one of them wins the race**, causing the other processor's copy to be invalidated. For the other processor to complete its write, it must obtain a new copy of the data, which now contains the updated value. Therefore, this protocol **enforces write serialisation**.

The alternative to an invalidate protocol is to update all the cached copies of a data item when that item is written. This type of protocol is called a **write update** or **write broadcast** protocol. Because a write update protocol must **broadcast all writes** to shared cache lines, it **consumes considerably more bandwidth**. For this reason, recent multiprocessors have opted to implement write invalidate protocols and we will only focus on those for the rest of this section.

Finding data in caches

In addition to invalidating outstanding copies of a cache block that is being written into, we also need to locate a data item when a cache miss occurs.

In a write-through cache, it's easy to find the recent value of a data item, since all written data are always sent to the memory, from which the most recent value of that data can always be fetched. For a write-back cache, the problem of finding the most recent data value is harder, since it may be in a private cache rather than in the shared cache memory.

Fortunately, write-back caches can use the **same snooping scheme** both for cache misses and writes: each processor snoops every address placed on the shared bus. If a processor finds that **it has a dirty copy of the requested block**, it **provides that cache block** in response to the read request and causes the **memory access to be aborted**.

Since write-back caches generate **lower requirements for memory bandwidth**, they can support larger numbers of faster processors. As a result, all multicore processors use **write-back caches at the outermost levels of the cache**.

The normal cache tags can be used to implement the process of snooping and the valid bit for each block makes invalidation easy to implement.

Read misses are also straightforward, since they simply rely on the snooping capability.

For writes, we would like to know whether any other copies of the block are cached because, if there are no other cached copies, then the write need not be placed on the bus in a write-back cache. To track whether or not a cache block is shared, we can **add an extra state bit** associated with each cache block, just as we have a valid and a dirty bit. By adding a bit indicating whether the block is shared, we can decide whether a write must generate an invalidate.

When a write to a block in the shared state occurs, the cache generates an invalidation on the bus and **marks the block as exclusive**. No further invalidations will be sent by that core for that block.

The core with the sole copy of a cache block is normally called the **owner** of such cache block.

When an invalidation is sent, the state of the owner's cache block is changed from shared to exclusive. If another processor later requests this cache block, the state must be made shared again.

MSI protocol

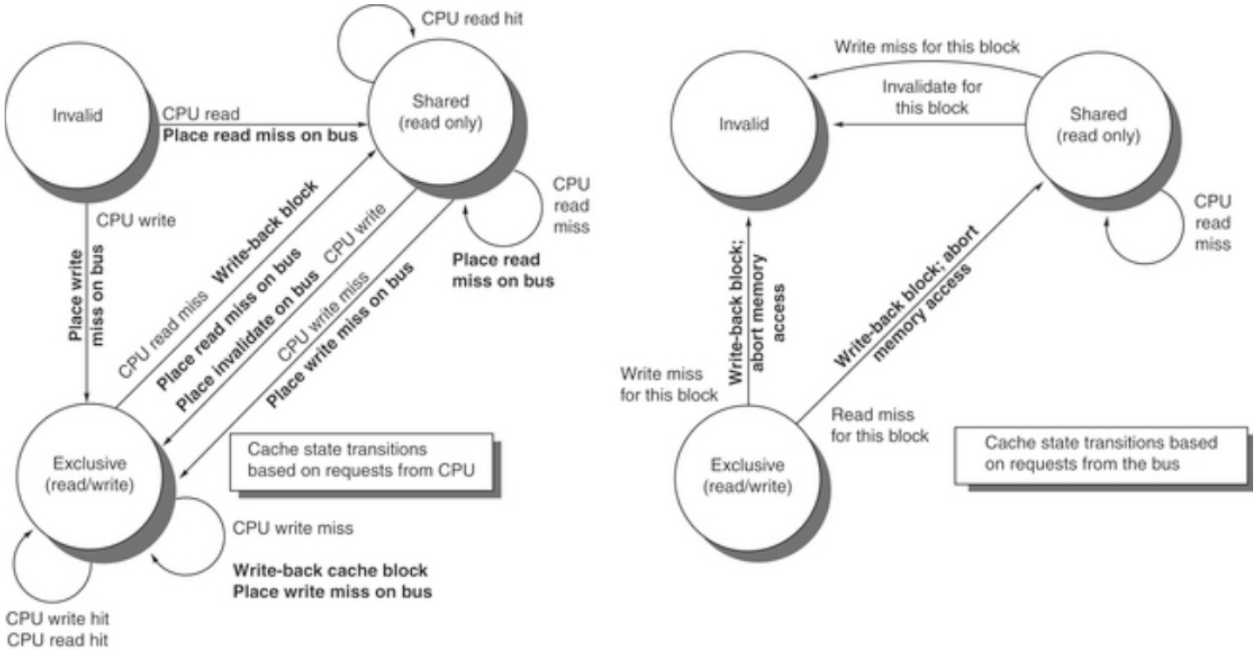
A snooping coherence protocol is usually implemented by incorporating a finite-state controller in each core. This controller responds to requests from the processor in the core and from the bus, changing the state of the selected cache block as well as using the bus to access data or to invalidate it.

We will now describe a simple protocol that uses only three states: invalid, shared and modified—usually called **MSI** protocol from the initials of its states.

The **shared** state indicates that the block in the private cache is potentially shared, while the **modified** state *implies* that the block is exclusive.

When an invalidate or a write miss is placed on the bus, any cores whose private caches have copies of the cache block invalidate it. For a write miss in a write-back cache, if the block is exclusive in just one private cache, that cache also writes back the block; otherwise, the data can be read from the shared cache or memory.

Below is a finite-state transition diagram that explains how this protocol works, distinguishing between processor requests (on the left) and bus requests (on the right):



MSI EXTENSIONS: THE MESI PROTOCOL

There are many extensions of the above mentioned protocol, which are created by adding additional states and transactions that optimise certain behaviours, possibly resulting in improved performance.

One of this extensions is the so-called **MESI** protocol, which adds the **exclusive** state to the original protocol to indicate when a cache block is resident only in a single cache but is clean.

If a block is in the exclusive state, **it can be written without generating any invalidates**, which optimises the case where a block is read by a single cache before being written by that same cache.

Of course, when a read miss to a block in the exclusive state occurs, the block must be changed to the shared state to maintain coherence.

Synchronisation problems

The major complication in actually implementing snooping coherence protocols is that write and upgrade misses **are not atomic** in any recent multiprocessor. The steps of detecting a write or upgrade miss, communicating with the other processors and memory, getting the most recent value for a write miss and ensuring that any invalidates are processed, and updating the cache **cannot be done as if they took a single clock cycle**.

In a single multicore chip, these steps can be made effectively atomic by arbitrating for the bus to the shared cache or memory first and not releasing the bus until all actions are complete. In some multicores, a single line is used to signal when all necessary invalidates have been received and are being processed.

In a system without a bus, we must find some other method of making the steps in a miss atomic. In particular, we must ensure that two processors that attempt to write the same block at the same time are **strictly ordered**.